Create a custom module

1. fetch the module source code from

A module is a reusable, standalone script that Ansible runs on your behalf, either locally or remotely. Modules interact with your local machine, an API, or a remote system to perform specific tasks like changing a database password or spinning up a cloud instance.
Each module can be used by the Ansible API, or by the ansible or ansible-playbook programs. A module provides a defined interface, accepts arguments, and returns information to Ansible by printing a JSON string to stdout before exiting.
We will create a new module in the next steps, its source code is already present in this git repository: https://github.com/ghazelatech/custom-module

2. Create a directory for your module

To begin, we'll establish a library directory at the root of our repository to house our custom module. By having a ./library directory relative to their YAML file, playbooks can incorporate custom Ansible modules that are then recognizable within the Ansible module path. This structure allows us to conveniently group custom modules and their corresponding playbooks.

3. Write the module code

The custom Python module **epoch_converter.py** needs to be created inside the **library** directory. This module converts an **input epoch_timestamp** argument to a **datetime** type. Additionally, it uses a **state_changed** argument to **simulate** a change in the target system by this module.

```python
#!/usr/bin/python

from __future__ import (absolute_import, division, print_function)
__metaclass__ = type
import datetime

DOCUMENTATION = r'''
---
module: epoch_converter

short_description: This module converts an epoch timestamp to human-readable date.

# If this is part of a collection, you need to use semantic versioning,
# i.e. the version is of the form "2.5.0" and not "2.4".
version_added: "1.0.0"
```

```
description: This module takes a string that represents a Unix epoch timestamp and displays its human-readable date
equivalent.

options:
  epoch_timestamp:
    description: This is the string that represents a Unix epoch timestamp.
    required: true
    type: str
  state_changed:
    description: This string simulates a modification of the target's state.
    required: false
    type: bool

author:
  - Ioannis Moustakis (@Imoustak)
'''

EXAMPLES = r'''
# Convert an epoch timestamp
- name: Convert an epoch timestamp
 epoch_converter:
  epoch_timestamp: 1657382362
'''

RETURN = r'''
# These are examples of possible return values, and in general should use other names for return values.
human_readable_date:
  description: The human-readable equivalent of the epoch timestamp input.
  type: str
  returned: always
  sample: '2022-07-09T17:59:22'
original_timestamp:
  description: The original epoch timestamp input.
  type: str
  returned: always
  sample: '16573823622'

'''

from ansible.module_utils.basic import AnsibleModule


def run_module():
  # define available arguments/parameters a user can pass to the module
  module_args = dict(
    epoch_timestamp=dict(type='str', required=True),
    state_changed=dict(type='bool', required=False)
  )

  # seed the result dict in the object
  # we primarily care about changed and state
  # changed is if this module effectively modified the target
  # state will include any data that you want your module to pass back
  # for consumption, for example, in a subsequent task
  result = dict(
    changed=False,
    human_readable_date='',
    original_timestamp=''
  )

  # the AnsibleModule object will be our abstraction working with Ansible
```

```
    # this includes instantiation, a couple of common attr would be the
    # args/params passed to the execution, as well as if the module
    # supports check mode
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    # if the user is working with this module in only check mode we do not
    # want to make any changes to the environment, just return the current
    # state with no modifications
    if module.check_mode:
        module.exit_json(**result)

    # manipulate or modify the state as needed (this is going to be the
    # part where your module will do what it needs to do)
    result['original_timestamp'] = module.params['epoch_timestamp']
    result['human_readable_date'] = datetime.datetime.fromtimestamp(int(module.params['epoch_timestamp']))

    # use whatever logic you need to determine whether or not this module
    # made any modifications to your target
    if module.params['state_changed']:
        result['changed'] = True

    # during the execution of the module, if there is an exception or a
    # conditional state that effectively causes a failure, run
    # AnsibleModule.fail_json() to pass in the message and the result
    if module.params['epoch_timestamp'] == 'fail':
        module.fail_json(msg='You requested this to fail', **result)

    # in the event of a successful module execution, you will want to
    # simple AnsibleModule.exit_json(), passing the key/value results
    module.exit_json(**result)


def main():
    run_module()


if __name__ == '__main__':
    main()
```

4. Test the new module

A playbook named **test_custom_module.yml** can be created in the same directory as our library directory to test the module.

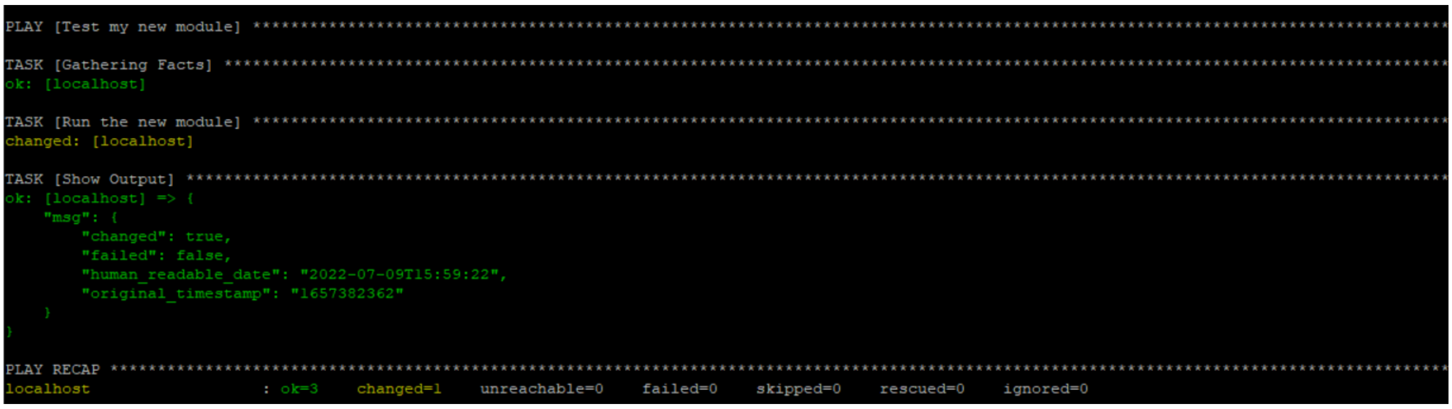
```
Python
- name: Test my new module
  hosts: localhost
  tasks:
  - name: Run the new module
    epoch_converter:
      epoch_timestamp: '1657382362'
```

```
        state_changed: yes
      register: show_output
    - name: Show Output
      debug:
        msg: '{{ show_output }}'
```

Let's execute the playbook to test our custom module:

```
$ ansible-playbook test_custom_module.yml -i localhost
```

The task state should be displayed in yellow, indicating it has changed, due to the state_changed argument being set.



```
PLAY [Test my new module] ******************************************************************************************

TASK [Gathering Facts] *********************************************************************************************
ok: [localhost]

TASK [Run the new module] ******************************************************************************************
changed: [localhost]

TASK [Show Output] *************************************************************************************************
ok: [localhost] => {
    "msg": {
        "changed": true,
        "failed": false,
        "human_readable_date": "2022-07-09T15:59:22",
        "original_timestamp": "1657382362"
    }
}

PLAY RECAP *********************************************************************************************************
localhost                  : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Ansible modules vs. plugins

Ansible comprises both modules and plugins, each serving distinct roles. Modules are units designed to execute specific actions on target hosts.
Examples include the 'file' module for file management and the 'apt' module for package handling on Debian-based systems. On the other hand, plugins operate on the control node, extending Ansible's core functionality by offering features like connection management, data lookups, and output filtering.

In essence, while modules carry out the tasks defined within Ansible playbooks, plugins enhance the overall Ansible environment and workflow. This enhancement can be either implicit or achieved through explicit configuration.

If you want to experiment the usage of custom plugin, you might want to check the example in this repository:
https://github.com/ghazelatech/custom-plugin/tree/main